Qt Quality Assurance

WHITEPAPER

Towards Autonomous Testing

How can we automate test case design?

Insights from Alexej Popovič and Dr. Maximilian Blochberger

Overview

Automated execution of tests as part of the continuous integration (CI) process is a well - established practice for efficiently discovering bugs early in the development process. However, tests are typically created manually. Identification and implementation of meaningful test cases significantly contributes to the total cost of the testing process. To reduce this cost and to take one step towards a fully autonomous testing process, test case design and implementation needs to be automated. Like exploratory testing, an autonomous tester interacts with the application under test (AUT) to discover its structure and generate optimized test cases. In this paper, we present ideas of how an autonomous system might operate and discuss challenges for creating such a system.

With software development gradually embracing agile principles and continuous integration and delivery methods, test automation has become an integral part of the product development lifecycle. By reducing human effort and speeding up the execution cycles, test automation enables shift-left testing, i.e., testing as early as possible, thereby lowering the cost of fixing defects [1].

However, the term test automation usually refers to test execution, while the definition and implementation of test cases are tasks carried out manually by testers. We will discuss how this could be automated as well.

Creating test cases depends on the available information about the inner workings and the expected behavior of the AUT. While black-box testing solely considers the system's input and expected outputs, white-box testing also considers how the system arrives at the outputs. On the one hand, white-box testing techniques already allow for autonomous testing in practice. There are tools available to generate (i) unit tests for a given function based on its source code[1], (ii) input data triggering certain conditions, such as crashes (fuzz testing), or (iii) the identification of common errors such as type mismatches, missing null-pointer checks, among others that can usually be identified statically, i.e., without executing the AUT.
For black-box testing, on the other hand, test cases are created manually based on requirements, specification, or by discovering potential test cases through exploratory testing. Automatic creation of test cases usually requires additional information for that purpose, such as models for model-based testing.

To further reduce the cost of the testing process, we are focusing on black-box test generation of GUI applications.

[1] CodiumAI (https://www.codium.ai/)
EvoSuite (https://www.evosuite.org/)
Parasoft Jtest (https://www.parasoft.com/jtest)
AgitarOne (http://www.agitar.com/solutions/products/agitarone.html)

Approach

When creating test cases for GUI applications, the specification of the AUT may be unavailable. In that case, a tester would typically utilize exploratory testing, which may be described as simultaneous learning, test design, and test execution [1,2]. Testers use their experience and intuition to systematically explore the AUT's features and treat what they learned as a specification, while also assessing the correctness of the AUT's behavior.

Our proposal for autonomous test case design and implementation is based on exploratory and monkey testing. The approach conceptually consists of four phases:

1. **Exploration**
2. **Generalization**
3. **Optimization**
4. **Validation**

These phases may be iterated until a pre-defined exit criterion is met. In practice, the four phases are intertwined, as illustrated in Figure 1, so that immediate feedback can influence which actions are performed.

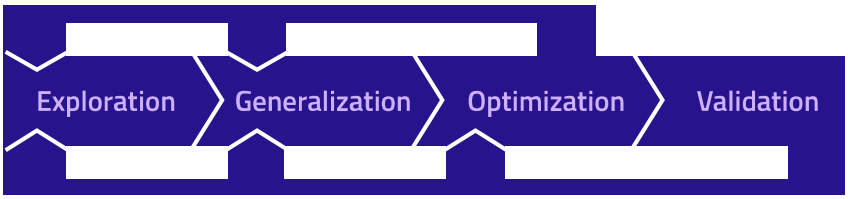

Figure 1: The four phases of autonomous test case design, implementation and execution.

To interact with the AUT, it needs to be executed. In the exploration phase, the autonomous tester then subsequently (i) takes a snapshot of the AUT's state, (ii) determines the next action, (iii) generates input data for that action, and (iv) performs the action. The states and actions are logged into a trace, as illustrated in Figure 2. A trace can later be replayed as a test case.

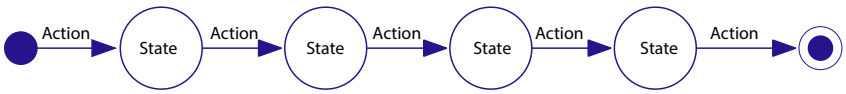

Figure 2: A trace collected during the exploration of an application.

In the generalization phase, a state machine (model) of the AUT is estimated based on the collected traces by merging states into abstract states [3], as illustrated in Figure 3 (see next page).

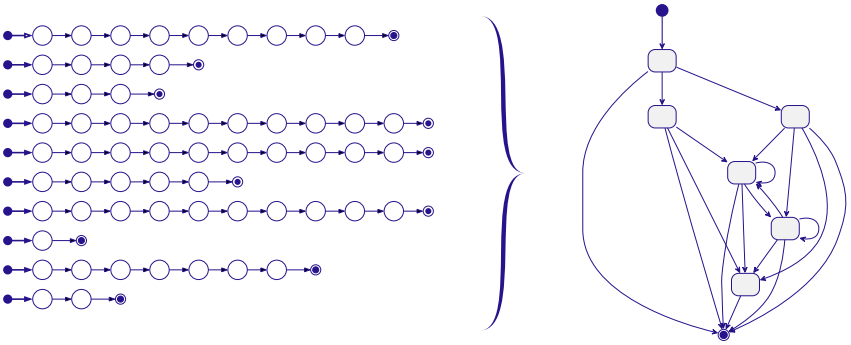

Figure 3: The generalization of a model based on multiple traces.

In the optimization phase, the derived model of the AUT is utilized to identify potential improvements to individual test cases, e.g., shorter paths to reproduce identified defects, or to omit irrelevant test cases.

In the validation phase, the correctness of the AUT is assessed. Note that this is limited to universal malfunctions, such as crashes, to heuristics, or to non-functional tests, such as usability issues. For functional tests, a specification needs to be provided as additional input. Further validation is performed on the detected test cases themselves, e.g., to identify whether they are reproducible or whether a minimized test case still reveals the originally detected defect.

The described approach already combines test case design and implementation with test execution. Since the AUT needs to be executed for discovery, there is no need to do so for validation again. The collected traces even contain sufficient information to enable offline testing.

Challenges

The approach presented in the last section omits many details for brevity. In this section, we discuss selected challenges for making the approach fully autonomous and outline potential solutions.

During exploration, we face challenges of black-box testing, such as unknown variables influencing an application's internal state, e.g., the state of the hardware it is running on, time, or sensor input. Furthermore, only a well-known set of actions can be considered at a time. Complex actions, such as entering specific texts or performing touch gestures are challenging to detect automatically. The available information about the AUT depends on the runtime introspection capabilities of the used toolkit or the operating system.

During generalization, the main challenge is to find a model that is abstract enough but still useful for generating test cases. Furthermore, the data considered by the similarity metric used for merging states into abstract states will largely depend on the application domain and the specific test goals.

During optimization, complexity presents the biggest challenge. If a crash was identified, first, the trace needs to be replayed to validate reproducibility, then all shorter paths leading to the crash based on the derived model need to be executed to validate whether the result of the optimization is still revealing the crash.

During validation, we mainly face the problem that the business logic of the AUT is unknown. As a person, you can intuitively conclude for a calculator the output 3 for the input 1+1 is a defect. A software system generally cannot assess the correctness of the output without the expectation being provided as well. AI/ML methods can be applied to predict the expected outcome, such an approach may lack traceability information, however. Furthermore, an autonomous tester can only validate what it can observe, i.e., it cannot detect missing features. Heuristics can be applied to detect general mistakes, e.g., looking for the term error in window titles [4], or identifying differences to a previous run as is done for visual GUI testing [5] and could help detecting features that were removed by accident.

Related work

Autonomous exploration of the AUT is usually known as monkey testing [6–9]. These can perform random actions to crawl the AUT or prioritize actions with a higher chance of discovering new states [10]. To achieve that, one approach has been Q-learning [3,10–12]. Some tools use Large Language Models (LLMs) to generate a formal description of a test case based on the product specification written in natural language. LLMs can also generate code based on natural language descriptions. Therefore, generating test scripts using the same specification may become a reality soon. Valdés et al. provide a comprehensive overview of studies related to GUI test automation.

Conclusion

Because there is no universal oracle that can decide whether a behavior is correct, we may never achieve full autonomy in testing. Consider autonomously testing the driving unit of a vehicle. If the autonomous tester were able to always tell if the car is driving correctly, always choosing the "correct" way to drive would thus solve the challenge of autonomous driving. Ad absurdum, generalizing to arbitrary systems, the autonomous tester would know everything, as it can tell apart correct and incorrect information. Nevertheless, even if the autonomous tester is limited to a set of selected use cases, it would still be valuable: finding ways to crash the AUT, recognizing wrong translations, inconsistencies both in behavior and in the interface, verifying usability and accessibility, or finding regressions. This allows testers to focus on validating the business logic. In this work, we only focused on one part of what needs to be done to achieve full autonomy in testing. The testing process involves more than executing applications and collecting results. We may also strive for autonomous test planning and result evaluation, possibly even including suggestions for fixes.

The findings of this whitepaper were presented at the ESE Kongress 2023.

References

[1] R. Patton, *Software testing*, 2nd ed. Indianapolis, IN: Sams Pub, 2006

[2] J. Bach, 'Exploratory Testing Explained', *Satisfice, Inc.*, 2003

[3] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, 'Automatic testing of GUI-based applications', *Softw. Test. Verification Reliab.*, vol. 24, no. 5, 2014

[4] S. Bauersfeld, T. E. J. Vos, N. Condori-Fernández, A. Bagnato, and E. Brosse, 'Evaluating the TESTAR tool in an industrial case study', in *ESEM*, ACM, 2014

[5] E. Börjesson and R. Feldt, 'Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry', in *ICST*, IEEE Computer Society, 2012

[6] A. M. Memon, I. Banerjee, and A. Nagarajan, 'GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing', in *WCRE*, IEEE Computer Society, 2003

[7] D. Amalfitano, A. R. Fasolino, and P. Tramontana, 'A GUI Crawling-Based Technique for Android Mobile Application Testing', in ICST Workshops, IEEE Computer Society, 2011

[8] 'UI/Application Exerciser Monkey', *Android Developers*. Accessed: Oct. 12, 2023. Available: https://developer.android.com/studio/test/other-testing-tools monkey

[9] N. Nyman, 'Using monkey test tools', *Software Testing & Quality Engineering*, 2000

[10] S. Bauersfeld and T. E. J. Vos, 'User Interface Level Testing with TESTAR; What about More Sophisticated Action Specification and Selection?', in *SATToSE*, in CEUR Workshop Proceedings, vol. 1354. CEUR-WS.org, 2014

[11] J. Eskonen, J. Kahles, and J. Reijonen, 'Automating GUI Testing with Image-Based Deep Reinforcement Learning', in *ACSOS*, IEEE, 2020

[12] D. Adamo, M. K. Khan, S. Koppula, and R. C. Bryce, 'Reinforcement learning for Android GUI testing', in *A-TEST@ESEC/SIGSOFT FSE*, ACM, 2018

[13] O. R. Valdés, T. E. J. Vos, P. Aho, and B. Marín, '30 Years of Automated GUI Testing: A Bibliometric Analysis', in *QUATIC*, in Communications in Computer and Information Science, vol. 1439. Springer, 2021

Authors

Alexej Popovič is a software engineer with a passion for knowledge representation and reasoning as well as robotics. He presently works at Qt Group, where he helps to develop tools for software testing.

Dr. Maximilian Blochberger is a software engineer at Qt Group and a security researcher. He is primarily interested in aiding developers to create secure and privacy-friendly applications, even if they lack the necessary background.

Contact us for more information.

Qt Quality Assurance

www.qt.io/squish

© Qt Group_20240925